Nearest string korišćenjem RVNS i Simuliranog kaljenja

Projekat iz Računarske Inteligencije
Matematički fakutet
Univerzitet u Beogradu

Sara Kapetinić
mi17182@alas.matf.bg.ac.rs

septembar 2022

Sadržaj

1 Opis problema **3**

2 Ulazni i test podaci **3**

3 RVNS – Uopšteno **4**

4 Simulirano kaljenje – Uopšteno **5**

5 Brute force **6**
5.1 Implementacija 6
6 Optimizacija – RVNS **8**
6.1 Uvod 8
6.2 Funkcija evaluacije 8
6.3 Konstrukcija početnog rešenja 8
6.4 Generisanje susedstva 9
6.5 Kompletno RVNS rešenje 10

7 Optimizacija – Simulirano kaljenje **11**
7.1 Uvod 11
7.2 Implementacija 11

8 Drugi optimizacioni algoritam **12**
8.1 Ideja 12
8.2 Implementacija 13

9 Testovi i rezultati **14**

10 Literatura **17**

1 Opis problema

Nearest string problem je problem kombinatorne optimizacije, koji kao ulaz prima skup stringova $S = \{s_1, s_2, ...s_n\}$ nad alfabetom Σ, gde je n broj stringova u skupu, koji su jednake dužine. Potrebno je pronaći string čija je distanca od datog skupa stringova minimalna. Za meru distance se uzima Hamingovo rastojanje izmedju dva stringa. Distanca od rešenja od najudaljenijeg stringa iz skupa se smatra objektivnom vrednošću rešenja koja je potrebna da se minimizuje. Prolaženje stringa koji zadovoljava prethodne uslove je NP-težak problem i ima dosta primena u realnim problemima. Najveću primenu ovog problema je u bioinformatici, u radu sa DNK sekvencama.

2 Ulazni i test podaci

Ulazni podaci su stringovi, koji se ucitavaju iz navedenog tekstualnog fajla. Stringovi su smešteni u std::vector. Alfabet koji se koristi u algoritmima su mala slova engleskog alfabeta.

Radi testiranja programa napravljena je klasa makeBigFile, koja generiše reči u fajl. Broj reči i njihova dužina se mogu zadati i time se pruža mogućnost testiranja algoritama na veoma veliki skupovima stringova. Najmanji skupovi na kojima su testirani algoritmi imaju 4 reči sačinjene od 3 karaktera(radi provere ispravnosti algoritama), dok su najveći skupovi generisani za testiranje imali 10000 stringova.

3 RVNS – Uopšteno

RVNS (Reduced Variable Neighborhood Search) je metaheuristika koja spada u algoritme lokalne pretrage i koristi se za rešavanje kombinatornih optimizacionih problema.

Razmotrimo kombinatorni ili globalni problem optimizacije

$$\min f(x)$$

gde je $x \in X$.

Funkcija $f(x)$ se naziva funkcija cilja koja se minimizuje, dok je X skup svih dopustivih rešenja. Rešenje $x^* \in X$ je optimalno ukoliko važi:

$$f(x^*) \leq f(x) , \forall x \in X ;$$

Glavna pitanja kod metaheurističkih pretraga su:

1. *U kom smeru nastaviti pretragu?*
2. *Koliko daleko?*
3. *Šta dalje, ukoliko potez nije zadovoljavajući?*

Ovo su ciljevi koji se obradjuju u RVNS pretrazi. Uzimamo u obzir skup suseda $N_1(x)$, $N_2(x)$ ... $N_{kmax}(x)$, gde je kmax maksimalna okolina trenutno rešenja x koju pretražujemo. Ukoliko smo u nekom od susedstva pronašli bolje rešenje od trenutnog, pretragu ćemo nastaviti od njega. U suprotnom, nastavljamo sa sledećim susedstvom. Nakon razmatranja svih susedstva, ponovo pocinjemo sa prvim, sve dok se ne zadovolji navedeni uslov zaustavljanja.

4 Simulirano kaljenje – Uopšteno

Algoritam simuliranog kaljenja (eng. *simulated annealing*, skraćeno SA) je zasnovan na procesu kaljenja čelika, čiji je cilj oplemenjivanje metala tako da on postane čvršći. Prvi korak u kaljenju čelika je zagrevanje do određene temperature, a zatim, nakon kratkog zadržavanja na toj temperaturi, počinje postepeno hladenje. Pritom treba voditi računa o brzini hlađenja, jer brzo hladenje može da uzrokuje pucanje metala.

Simulirano kaljenje se zasniva na poboljšavanju vrednosti jednog rešenja. Na početku algoritma se proizvoljno ili na neki drugi način generiše početno rešenje i izračuna vrednost njegove funkcije cilja. Vrednost najboljeg rešenja se najpre inicijalizuje na vrednost početnog. Zatim se algoritam ponavlja kroz nekoliko iteracija. U svakom koraku se razmatra rešenje u okolini trenutnog. Ukoliko je vrednost njegove funkcije cilja bolja od vrednosti funkcije cilja trenutnog rešenja, ažurira se trenutno rešenje. Ukoliko vrednost funkcije cilja novog rešenja nije bolja od vrednosti funkcije cilja trenutnog, upoređuju se vrednosti unapred definisane funkcije p i proizvoljno izabrane vrednosti q iz intervala (0,1). Ako je $p > q$, trenutno rešenje se ažurira novoizabranim. Takođe se, po potrebi, ažurira i vrednost najboljeg dostignutog rešenja. Algoritam se ponavlja dok nije ispunjen kriterijum zaustavljanja.

Drugi naziv za simulirano kaljenje je Monte Carlo kaljenje ili stohastičko kaljenje.

5 Brute force

5.1 Implementacija

Implementacija brute force algoritma se zasniva na iterativnom pristupu i generisanje permutacija. Funkcija generate_words generišu se sve permutacije, dužine m, gde je m dužina stringova u inicijalnom skupu. Sve permutacije se smeštaju u std::vector<std::string>. U slučaju da se u skupu stringova sa ulaza ne nalaze svi karakteri iz Σ, redukuje se alfabet na skup karaktera koji se pojavljuju u ulaznom skupu. U petlji se prolazi kroz sve generisano permutacije i pokreće funkcija findMaximumHammingDistance. Ova funkcija prima string i skup stringova i vraća maksimalno Hamingovo rastojanje. Hamingovo rastojanje izmedju dva stringa je broj karaktera u kojima se oni razlikuju. Ukoliko funkcija findMaximumHammingDistance vrati vrednost koja je manja od trenutno zapamćene, ažuriramo trenutno najbolju vrednost i pamtimo trenutnu permutaciju.

Algorithm 1 Haming distance

Input: *String* s_1, *String* s_2
Output: Broj mesta na kojima se razlikuju ulazni stringovi
*broja*č = 0
for $i \leftarrow 1$ *to* s_1*.size do*
    Ako se karakteri $s_1[i]$ *i* $s_2[i]$ razlikuju then
        *broja*č ++
return *broja*č

Algorithm 2 findMaximumHammingDistance

Input: *String permutation, vector<string> S*
Output: Maksimalno Hamingovo rastojanje izmedju permutacije i vectora stringova

```
bestDistance
for s in S do
    izračunaj Hamingovo rastojanje izmedju permutation i s i pamti kao
    trenutno
    if (trenutno_rastojanje > bestDistance)
        bestDistance ← trenutno_rastojanje
return  bestDistance
```

Algorithm 3 Brute Force

Input: *vector<string> permutacije, vector<string> S*
Output: string koji minimizuje funkciju cilja findMaximumHammingDistance

```
bestDistance
Solution
for p in permutacije do
    izračunaj najveće Hamingovo rastojanje izmedju p i S
    if (trenutno_rastojanje < bestDistance)
        bestDistance ← trenutno_rastojanje
        solution  ← p
return  solution
```

Složenost ovog algoritma nosi generisanje permutacija, čija je slozenost eksponencijalna. Postoje razni načini za mala poboljšavanja ovog algoritma. Recimo, pri traženju maksimalnog Haming rastojanja, u slučaju da ranije nadjemo Hamingovo rastojanje u dužine stringova mozemo prekinuti pretragu i vratiti to kao dužinu stringa kao povratnu vrednost.

6 Optimizacija – RVNS

6.1 Uvod

Kako je i navedeno u objašnjenju RVNS algoritma ova metaheuristika se koristi za probleme kombinatorne optimizacije, kakav je i problem Nearest string. Nadalje objasnićemo pristup i način implementacije ovog algoritma.

6.2 Funkcija evaluacije

Kao funkciju evaluacije koristimo funkciju navedenu u prethodnom poglavlju o Brute force algoritmu.

$$f(solution,S) = max\ Dist(solution,s_i)\ ,\ i = 1\ ..\ n$$

gde je solution rešenje, S je skup stringova, a n je broj stringova u skupu S. Funkcija Dist vraća Hamingovo rastojanje izmedju 2 stringa. Pseudo kod za navedene funkcije nalazi se u prethodnom poglavlju(*Str:6 i Str:7*).

6.3 Konstrukcija početnog rešenja

Prvi korak kod implementacije algoritma je konstrukcija početnog rešenja. Počinjemo od alfabeta i broja karaktera za inicijalizaciju rešenja. Funkcija *initialize* prima ove vrednosti kao argumente, a povratna vrednost je string koji se uzima kao početno rešenje.

Implementacija funkcije initialize se svodi na generisanje random karaktera. Ukoliko sa *m* obeležimo dužinu stringa koji treba da generišemo, problem se svodi na generisanje *m* slučajnih karaktera iz alfabeta koji primamo kao argument funkcije. Konstrukcija stringa se zasniva na konkatenaciji slučajno odabranih karaktera.

Algorithm 4 Initialize

Input: *Alfabet Σ,* dužina k
Output: string konstruisan od k slučajnih karaktera iz alfabeta Σ
for i ← 1 to k do
    Generišemo slučajne indekse iz uniformne raspodele (0,veličina alfabeta)
for *indeks in generisani_indekse* do
    Dodajemo karakter iz alfabeta na poziciji indeks na solution
return *solution*

6.4 Generisanje susedstva

Koncept algoritma je pretraga susedstva. Funkcija getNeighbour kao argumente prima *solution* i veličinu susedstva *k.* Povratna vrednost funkcije je random generisan sused iz okoline k. Sused je iz okoline k ukoliko se razlikuje na k mesta od suseda.

Algoritam se svodi na generisanje slučajnih k indeksa u *solution*, kao i generisanje slučajnih k karaktera. *Solution* menjamo na odabranim indeksima sa generisanim karakterima. Uz to je potrebno da pamtimo indekse i prethodne karaktere u *solution* na tim pozicijama, što će biti objašnjeno u sledećem delu.

Algorithm 5 getNeighbour

Input: *Solution string,* susedstvo k
Output: *string iz k-susedstva stringa solution*
indeksi ← *generiši k* indeksa iz uniformne raspodele (0, dužina solution)
karakteri ← *generiši k* karaktera iz alfabeta Σ
prethodni_karakteri

for i ← 1 to k do
    u *prethodni_karakteri* dodaj karakter na poziciji *solution[indeks[i]]*
    postavi na *solution[indeks[i]]* karakter iz skupa karakteri na poziciji *i*

return *solution*

6.5 Kompletno RVNS rešenje

Sada imamo skoro sve potrebne elemente za formiraje kompletnog rešenja. Poslednja funkcija vraća solution u prethodno stanje, pre izmene kod obilazaka susedstva.

Algorithm 6 RVNS

Input: *Max_iters, max_k*
Output: *string Solution*
inicijalizuj početno rešenje solution
pamti trenutnu vrednost funkcije evaluacije
postavi rezultat na solution
for i ← 1 to *M*ax_iters do
    k = 0
    while k ≤ *max_k* do
        pronadji suseda iz susedstva k
        izračunaj novu vrednost funkcije evaluacije
        if nova_vrednost < trenutne_vrednosti do
            nastavi pretragu od njega
            zapamti suseda u rezultat
        else do
            proširi susedstvo k+=1
            vrati *solution* na staru vrednost

return *rezultat*

7 Optimizacija – Simulirano kaljenje

7.1 Uvod

Radi testiranja i uporedjivanja sa RVNS algoritmom razvijen je i naredni algoritam gde koristimo Simulirano kaljenje. Funkcija evaluacije koja se koristi je ista kao i u prethodnom algoritmu, kao i konstrukcija početnog rešenja. Dodatne funkcije koje se koriste u ovom algoritmu koje se razlikuju u odnosu na prethodni su:

1. *invertSolution* – ova funkcija vraća string koji predstavlja solution sa malim izmenama
2. *oldSolution* – vraća solution na prethodno stanje

Kako smo mogli da vidimo, simulirano kaljenje u svakoj iteraciji pravi male promene trenutnog rešenja. Ukoliko smo naišli na bolje rešenje, treba nastaviti sa njim, u suprotnom se oslanjamo na teoriju verovatnoće, koja će odrediti da li nastavljamo sa novim ili starim rešenjem.

7.2 Implementacija

Algorithm 6 RVNS

Input: *Max_iters, max_k*
Output: *string Solution*
inicijalizuj početno rešenje solution
pamti trenutnu vrednost funkcije evaluacije
postavi rezultat na solution
for i ← 1 to *M*ax_iters do
    malo izmeni rešenje
    izračunaj novu vrednost funkcije evaluacije
    if nova_vrednost < trenutne_vrednosti do
        nastavi pretragu od njega

```
                    if nova_vrednost < najbolja_vrednosti do
                            najbolja_vrednost = nova_vrednost
                            zapamti izmenjeno rešenje u rezultat

                    else do
                            generišemo q iz U(0,1)
                            if  0.5 > q
                                    nastavljamo sa izmenjenim
                            else do
                                    solution na staru vrednost
return  rezultat
```

8 Drugi optimizacioni algoritam

8.1 Ideja

Osnovni koncept ovog algoritma je redjanje skupa stringova jedan ispod drugog, tako da formiraju matricu. U *i-toj* vrsti se nalazi string na *i-toj* poziciji u skupu, dok se u *j-toj* koloni nalaze karakteri na *j-toj* poziciji u stringovima. Ideja se zasniva na prebrojavanju pojavljivanja karaktera na svakoj od pozicija u stringovima. Funkcija evaluacije je ista kao i u prethodnim algoritmima, zasnovana na Hamingovom rastojanju izmedju stringova. Rezultujući string se dobija konkatenacijom odabranih karaktera na svakoj od pozicija, metod za odabir karaktera je broj pojavljivanja karaktera na datoj poziciji. Potrebno je maksimizovati ovaj metod.

8.2 Implementacija

Pravimo strukturu podataka koja će čuvati broj pojavljivanja svakog karaktera na datoj poziciji (jedan od načina implementacije je map<char,int> koja pamtim karakter i broj pojavljivanja). Za svaku poziciju čuvamo mapu. Prolazimo kroz sve stringove iz skupa stringova na ulazu i za svaki karakter pamtimo njegov broj pojavljivanja.

U petlji prolazimo kroz sve mape (po jedna za svaku poziciju) i karakter koji se preslikava u najveću vrednost nadovezujemo na rezultat.

Algorithm 7 Optimizacija na osnovu istraživanja

Input: *Skup stringova S*
Output: *string koji zadovoljava uslove problema*

```
for i ← 1 to dužina_stringova_u_S do
        mapa ← (karakter, broj_pojavljivanja)

for i ← 1 to dužina_stringova_u_S do
        for (karakter,broj) in mapa[i] do
                if broj>trenutni_najveci do
                        zapamti karakter
        dodajemo karakter u rezultat
return rezultat
```

9 Testovi i rezultati

Radi sporovodjenja testova i uporedjivanja rezultata, napravljena je klasa koja generiše stringove u fajl. Pomoću nje je moguće napraviti fajl sa proizvoljnim brojem stringova, koji su proizvoljne dužine. Skup stringova se čita iz fajla sa putanje koju prosledimo. Algoritmi su isprobani nad random rečima sastavljenih od engleskog alfabeta, a radi testiranja korektnosti generisani su posebni manunelno napisani fajlovi.

9.1 Rezultati

Broj karaktera	Broj stringova u skupu	Vreme izvršavanja (ms)	Tačnost algoritma	Tip podataka
3	10	6	Pass	Manuelan
4	10	55	Pass	Manuelan
3	1000	806	Pass	Random
4	1000	21504	Pass	Random
5	10	319	Pass	Manuel
6	10	3021	Pass	Manuel
6	1000	?	bad_alloc	Random

1.1 Brute force algoritam

Veliki problem ovog algoritma je generisanje permutacija nad velikim alfabetom. Vidimo da u slučaju reči od 6 karaktera algoritam se izvršava kada je alfabet redukovan (generisanje permutacija samo nad karakterima koji se pojavljuju u rečima, a ne celim alfabetom). U suprotnom, kad smo generisali veliki broj reči, gde je neminovno da se pojavi veliki broj karaktera iz celog alfabeta, dobili smo *exception:bad_alloc.* Algoritam uvek pronalazi tačan rezultat medjutim vreme izvršavanja jako brzo raste. Samo generisanje permutacija je eksponencijalne složenosti.

Broj karaktera	Broj stringova u skupu	Vreme izvršavanja (ms)	Tačnost algoritma	Tip podataka	Broj iteracija
3	10	136	Pass	Manuelan	105
4	10	1255	Pass	Manuelan	2444
5	10	65533	Pass	Manuelan	68590
6	10	130585	Pass	Manuelan	161989
3	1000	5519	Pass	Random	0
100	1000	17293	?	Random	0

1.2 RVNS algoritam

Primetimo da na malim ulaznim podacima, brute force radi brzo u odnosu na RVNS algoritam. Razlog je u broju iteracija koje dosta usporavaju izvršavanje algoritma. Nad random podacima resenje je nadjeno u prvoj iteraciji, zbog velikog alfabeta i velikog broja ulaznih stringova, pa samim tim je najbolja opcija ujedno i najgora. Poslednji test je pokrenut da bi prikazao koliko broj karaktera u rečima usporava izvršavanje programa. Ovaj algoritam nije izazivao nikakve izuzetke, jer je prostorna slozenost zadovoljavajuća.

Broj karaktera	Broj stringova u skupu	Vreme izvršavanja (ms)	Tačnost algoritma	Tip podataka	Broj iteracija
3	10	528	Pass	Manuelan	96
4	10	2993	Pass	Manuelan	19383
5	10	2654	Pass	Manuelan	22569
6	10	264314	Not optimal	Manuelan	31919
3	1000	1928	Pass	Random	0
100	1000	2764	?	Random	0

1.3 Simulirano kaljenje

Simulirano kaljenje je pokazalo bolje rezultate kad su u pitanju mali skupovi ulaznih podataka. Medjutim, za test podatke gde je broj karaktera 6+, nalaženje optimalnog rešenje je zahtevalo veliki broj iteracija. Poslednja dva testa su uradjena radi testiranja brzine algoritma na skupovima velikih podataka. Vidimo da dužina stringova nije znatno uticala na vremensku složenost. Algoritam je podržao velike skupove podataka bez problema alokacije.

Broj karaktera	Broj stringova u skupu	Vreme izvršavanja (ms)	Tačnost algoritma	Tip podataka
3	10	0	Pass	Manuelan
4	10	0	Pass	Manuelan
3	1000	0	Pass	Random
4	1000	1	Pass	Random
5	10	0	Pass	Manuel
6	10	0	Pass	Manuel
6	1000	1	Pass	Random

1.4 Drugi optimizacioni algoritam

Ovaj algoritam je pokazao veoma zadovoljavajuće rezultate. Brzo se izvršava za razne veličine ulaznih vrednosti. Random generisani podaci su nad velikim alfabetom, stoga su rezultati testiranja primenljivi za uporedjivanje vremena izvršavanja.

10 Literatura

[1] Bin Ma; Xiaming Sun (2008). "More Efficient Algorithms for Closest String and Substring Problems" *Research in Computational Molecular Biology*

[2] L. Gasieniec, J. Jansson, and A. Lingas. Efficient approximation algorithms for the hamming center problem. pages 905–906, 1999.

[3] Laurent Bulteau ∗ Falk Hüffner† Christian Komusiewicz Rolf Niedermeier, Multivariate Algorithmics for NP-Hard String Problems